Міністерство освіти України

Національний технічний університет "ХПІ"

кафедра "Стратегічного управління"

|  |  |
| --- | --- |
|  |  |

**Звіт до лабораторної роботи №4**

з дисципліни "Алгоритми на структури даних"

Виконав: студент групи КН-321А

Бородай Д. А.

Перевірив: старший викладач

Мошко Є.О.

Харків 2022

**Тема лабораторної роботи.** Алгоритми сортування асимптотичний аналіз алгоритмів.

**Мета:** вивчити основні алгоритми сортування масивів і освоїти їх на практиці. Перевірити роботу алгоритмів на різних наборах даних, провести асимптотичний аналіз алгоритмів сортування.

**Порядок виконання роботи**:

1. Створити функцію сортування масиву алгоритмами сортування, які треба обрати у таблиці 3 відповідно до свого варіанту. Варіант завдання вибирається за номером студента в списку групи.

2. Будь-яким способом заповнити елементи масиву значеннями.

3. Виконати сортування масиву першим алгоритмом і проаналізувати отримані результати.

4. Перевірити всі варіанти вихідного заповнення масиву: випадковим чином, відсортованого у порядку зростання, відсортованого у порядку спадання. Переконатися в правильності сортування у всіх випадках. Зробити висновки.

5. Повторити пункти 2-4 для другого та третього алгоритму сортування.

6. Дослідити складність алгоритмів. Провести асимптотичний аналіз алгоритмів сортування та зробити висновки.

7. Здійснити порівняння алгоритмів сортування для n = 10, 100, 1000, 10000 і в наступному порядку вхідних елементів:

−елементи вже впорядковані;

−елементи в зворотному порядку;

−розстановка елементів випадкова.

8. Виконання порівняння алгоритмів сортування на основі

наступних кроків:

−створити таблицю асимптотичних оцінок трудомісткості

алгоритмів в кращому, середньому, гіршому випадках;

−розставити лічильники операцій у функціях угруповань;

−провести експеримент, визначити середню кількість операцій

для різних угруповань, побудувати графіки;

−для кожного виклику сортування генерувати новий масив;

−створити таблиці і представити графіки експериментальних оцінок алгоритмів.

## ОПИС АЛГОРИТІМ СОРТУВАНЬ

**1.1Алгоритм сортування бульбашкою.**

Алгоритм полягає в циклічних проходах по масиву, за кожен прохід елементи масиву попарно порівнюються і, якщо їх порядок неправильний, то здійснюється обмін. Обхід масиву повторюється до тих пір, поки масив не буде впорядкований. Код даного алгоритму продемонстровано нижче.

void BubbleSort(int\* array, int size)

{

bool changeFlag = true;

int i = 1;

while (changeFlag){

changeFlag = false;

for (int j = size - 1; j >= i; j--)

if (array[j] < array[j - 1]) {

swap(array[j], array[j - 1]);

changeFlag = true;

}

i++;

}

}

Повний код програми наведено у додатку А.

У даному випадку алгоритм сортування бульбашкою трішки модифікований, тобто якщо в циклі вже була заміна, то на цьому цикл завершується.

**1.2 Алгоритм Шейкер сортування**

Сортування змішуванням (*cocktail sort, shaker sort*), або шейкерне сортування - вдосконалений різновид сортування методом бульбашки, при якому сортування проводиться у двох напрямках, змінюючи напрям при кожному проході.

Аналізуючи алгоритм сортування бульбашкою, можна помітити:

* якщо при обході частини масиву не відбувається обмін елементів, то цю область можна виключити, так як вона вже відсортована;
* при проході від кінця масиву до початку, мінімальне значення зміщується на першу позицію, при цьому максимальний елемент переміщається тільки на один індекс вправо.

Виходячи з наведених ідей, модифікуємо сортування бульбашкою наступним чином:

* на кожній ітерації, фіксуємо межі області масиву в якій відбувається обмін;
* пробігаємо масив по черзі від початку до кінця, та від кінця до початку.

Таким чином, ми переміщаємо мінімальний елемент на початок масиву, а максимальний - в кінець, після відкидаємо індекси цих елементів, цим самим - зменшуємо робочу область. Таким чином, за кожну ітерацію невідсортована частина зменшується на два елементи.

Приклад алгоритму Шейкер сортування наведено нижче.

void ShekerSort(int\* array, int size)

{

int left = 0, right = size - 1;

int flag = 1;

while ((left < right) && flag > 0)

{

flag = 0;

for (int i = left; i < right; i++)

{

if (array[i] > array[i + 1])

{

swap(array[i], array[i + 1]);

flag = 1;

}

}

right--;

for (int i = right; i > left; i--)

{

if (array[i - 1] > array[i])

{

swap(array[i], array[i - 1

flag = 1;

}

}

left++;

}

}

Повний код програми наведено у додатку А.

**1.3 Алгоритм блочного сортування**

Ідея алгоритму полягає в тому, щоб розбити інтервал[0, 1) на n однакових відрізків(кошиків), та розділити по цих кошиках n вхідних величин. Оскільки вхідні числа рівномірно розподілені, передбачається, що у кожному кошику потрапить невелика кількість чисел. Потім послідовно сортуються числа у кошиках. Відсортований масив виходить шляхом послідовного перерахування елементів кожного кошика. Алгоритмів з лінійним часом виконання O(N) (на вдалих вхідних даних). Приклад коду наведено нижче.

void BucketSort(int\* array, int size)

{

int minValue = array[0];

int maxValue = array[0];

for (int i = 1; i < size; i++)

{

if (array[i] > maxValue) {

maxValue = array[i];

if (array[i] < minValue) {

minValue = array[i];

}

int bucketLength = maxValue - minValue + 1;

vector<int>\* bucket = new vector<int>[bucketLength];

for (int i = 0; i < bucketLength; i++)

bucket[i] = vector<int>();

for (int i = 0; i < size; i++)

bucket[array[i] - minValue].push\_back(array[i]);

int k = 0;

for (int i = 0; i < bucketLength; i++)

{

int bucketSize = bucket[i].size();

if (bucketSize > 0)

{

for (int j = 0; j < bucketSize; j++)

{

array[k] = bucket[i][j];

k++;

}

}

}

}

Повний текст програми наведено у додатку А.

Недоліки: непригідний при великій кількості мало відмінних елементів, або ж на невдалій функції отримання номера кошика за вмістом елемента.

## АСИМПОТИЧНИЙ АНАЛІЗ

* 1. **Складність алгоритмів**

Як видно, алгоритм досить простий, але, як іноді зауважують, він є неперевершеним у своїй неефективності.



Рисунок 2.1 – Приклад бульбашкового сортування масиву

Цей алгоритм можна оптимізувати: у прикладі останні проходи на сортування не впливають. У такому випадку потрібно запам’ятати, чи здійснювався на даному проході обмін, якщо ні, алгоритм завершити. Можна також запам’ятовувати місце останнього обміну, щоб не рухатися до раніше встановленої межі, якщо в цьому немає необхідності (тобто, кожний наступний прохід починати не з початку (j = 1), а з того місця, де на попередньому проході відбувся перший обмін).

Крім того, слід звернути увагу на таку ситуацію:

Якісно інше поліпшення алгоритму можна отримати з наступного спостереження. Хоча легка бульбашка знизу підніметься вгору за один прохід, важкі бульбашки опускаються із найменшою швидкістю: один крок за ітерацію. Так що масив:

2 3 4 5 6 1

буде відсортований за 1 прохід, а сортування послідовності:

6 1 2 3 4 5

вимагає 5 проходів.

Щоб оптимізувати впорядкування даних, можна змінювати напрямок проходів почергово. Одержаний алгоритм називають "шейкер-сортуванням".

У найкращому випадку кількість проходів по масиву буде n-1. У випадку, якщо мінімальний елемент стоїть в кінці масиву, нам знадобиться n-1 операцій перестановки елементів, для того щоб мінімальний елемент потрапив на перше місце в масиві.

Кількість порівнянь при використанні цього методу:

.

Кількість обмінів:

.

Наведений нижче алгоритм Шейкер-сортування є вдосконаленим варіантом сортування простим обміном і має також назву метод “бульбашки з обмеженнями”.

Суть її полягає в тому, що напрями переглядів чергують: за проходом до кінця множини слідує прохід від кінця до початку вхідної множини. При перегляді в прямому напрямку запис з найбільшим ключем ставиться на своє місце в послідовності, при перегляді у зворотному напрямі – запис з самим меншим, як показано на рисунку 2.2.

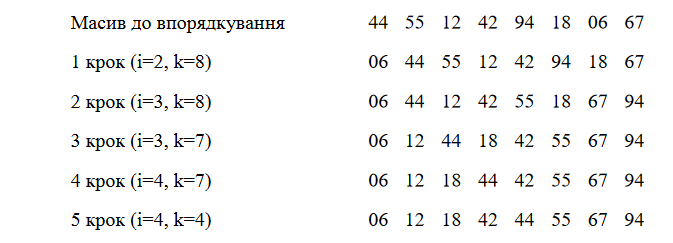


Рисунок 2.2 – Приклад шейкер-сортування

Цей алгоритм досить ефективний для задач відновлення впорядкованості, коли початкова послідовність вже була впорядкована, але піддалася не дуже значним змінам. Впорядкованість в послідовності з одиночною зміною (6 1 2 3 4 5) буде гарантовано відновлена усього за два проходи. Обмін двох елементів – набагато більш коштовна операція, ніж порівняння ключів, а при Шейкер-сортуванні зменшилась кількість порівнянь, але не обмінів.

Вважається, що бульбашкове та шейкер працює повільніше, ніж сортування вставками та вибором. Шейкер-сортування вигідно використовувати, якщо елементи масиву майже впорядковані.

Для шейкер-методу аналіз зробити складно. Але цей метод має сильну перевагу якщо масив "майже відсортовано", тобто, коли кількість елементів m, що не стоять на місці, є незначною у порівнянні з n, n << m. Тоді їх буде розставлено рівно за m проходів.

Що до складності та характеристики блочного сортування, то тут дещо інакше, оскільки метод сортування не схожий до попередніх.

**У Найгіршій складності,** коли в масиві є елементи з близьким діапазоном, вони, ймовірно, будуть розміщені в одному сегменті. Це може призвести до того, що в одних сегментах буде більше елементів, ніж у інших. Це робить складність залежною від алгоритму сортування, що використовується для сортування елементів сегмента. Складність стає ще гіршою, коли елементи знаходяться в зворотному порядку. Якщо для сортування елементів сегмента використовується вставка сортування, то складність часу стає меншою .

**Що до складності найкращого випадку:** Це відбувається, коли елементи рівномірно розподіляються в сегментах з майже рівною кількістю елементів у кожному сегменті. Складність стає ще кращою, якщо елементи всередині відрів вже відсортовані. Якщо для сортування елементів сегмента використовується вставка сортування, то в кращому випадку загальна складність буде лінійною, тобто. . )– це складність виготовлення сегментів і є складність сортування елементів сегмента з використанням алгоритмів, що мають лінійну часову складність у найкращому випадку.

**У середній складності –** . Це відбувається, коли елементи розподіляються хаотично в масиві. Навіть якщо елементи розподілені не рівномірно, сортування в сегменті виконується за лінійний час. Це справедливо до тих пір, поки сума квадратів розмірів відра не буде лінійною у загальній кількості елементів.

Таблица 2.1. Порівняння методів сортування

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | Best | Average | Worst | Memory | Stable |
| Bubble |  |  |  | 1 | Yes |
| Bucket |  |  |  |  | Yes |
| Shaker |  |  |  | 1 | Yes |

**2.2 Практичне порівняння**

Порівняння алгоритмів, описаних вище на масивах з кількістю елементів 10, 100, 1000, 10000. Усі вхідні данні однакові. Вигляд масив буде представлений у трьох варіаціях: випадково заповнений масив, впорядкований та у порядку спадання. Нижче наведено графіки та таблиці порівнянь, згідно розміру і типу масивів за кількістю операцій.

Таблиця 2.1 – Кількість операцій для випадкового масиву

|  |  |  |  |
| --- | --- | --- | --- |
| Sort/Size | Bubble sort | Shaker Sort | Bucket Sort |
| 10 | 73 | 72 | 535 |
| 100 | 7487 | 6556 | 903 |
| 1000 | 747089 | 632870 | 3611 |
| 10000 | 74937596 | 62797481 | 30611 |

Діаграма 2.1 – Кількість операцій для випадкового масиву

Таблиця 2.2 – Кількість операцій для відсортованого масиву

|  |  |  |  |
| --- | --- | --- | --- |
| Sort/Size | Bubble sort | Shaker Sort | Bucket Sort |
| 10 | 9 | 17 | 58 |
| 100 | 99 | 197 | 598 |
| 1000 | 999 | 1997 | 5998 |
| 10000 | 9999 | 19997 | 59998 |

Діаграма 2.2 – Кількість операцій для відсортованого масиву

Таблиця 2.3 – Кількість операцій для масиву за спаданням

|  |  |  |  |
| --- | --- | --- | --- |
| Sort/Size | Bubble sort | Shaker Sort | Bucket Sort |
| 10 | 90 | 90 | 58 |
| 100 | 9900 | 9900 | 598 |
| 1000 | 999000 | 999000 | 5998 |
| 10000 | 99990000 | 99990000 | 59998 |

Діаграма 2.1 – Кількість операцій для випадкового масиву

# Висновок

# Додаток А

Код Source.cpp

# Додаток Б

Код main.cpp

# Додаток В

Код Header.h